A Framework for Scientific Workflow Reproducibility in the Cloud

Rawaa Qasha
Newcastle University
Newcastle upon Tyne, UK
Mosul University, Iraq
Email: r.qasha@newcastle.ac.uk

Jacek Cała
Newcastle University
Newcastle upon Tyne, UK
Email: jacek.cala@newcastle.ac.uk

Paul Watson
Newcastle University
Newcastle upon Tyne, UK
Email: paul.watson@newcastle.ac.uk

***Abstract*—Workflow is a well-established means by which to capture scientific methods in an abstract graph of interrelated processing tasks. The reproducibility of scientific workflows is therefore fundamental to reproducible e-Science. However, the ability to record all the required details so as to make a workflow fully reproducible is a long-standing problem that is very difficult to solve.**

In this paper, we introduce an approach that integrates system description, source control, container management and automatic deployment techniques to facilitate workflow reproducibility. We have developed a framework that leverages this integration to support workflow execution, re-execution and reproducibility in the cloud and in a personal computing environment.

We demonstrate the effectiveness of our approach by examining various aspects of repeatability and reproducibility on real scientific workflows. The framework allows workflow and task images to be captured automatically, which improves not only repeatability but also runtime performance. It also gives workflows portability across different cloud environments. Finally, the framework can also track changes in the development of tasks and workflows to protect them from unintentional failures.

I. Introduction

Workflows have become a valuable mechanism for specifying and automating scientific experiments running on distributed computing infrastructure. Researchers in different disciplines have embraced them to conduct a wide range of analyses and scientific pipelines [1], mainly because a workflow can be considered as a model defining the structure of the computational and/or data processing tasks necessary for the management of a scientific process [2].

However, workflows are not only useful in representing and managing the computation but also as a way of sharing knowledge and experimental methods. When shared, they can help users to understand the overall experiment, or they can become an essential building block in their new experiments. Lastly, workflows can also be used to repeat or reproduce the experiment and replicate the original results [3].

One of the major challenges in achieving workflow reproducibility, however, is the heterogeneity of workflow components which demand different, sometimes conflicting sets of dependencies. Ensuring successful reproducibility of workflows requires more than simply sharing their specifications. It also depends on the ability to isolate necessary and sufficient computational artifacts and preserve them with adequate description for future re-use [4].

A number of analyses and research efforts have already been conducted to determine the salient issues and challenges in workflow reproducibility [5], [6], [7], [8]. In short the issues can be summarized as: insufficient and non-portable description of a workflow including missing details of the processing tools and execution environment, unavailable execution environments, missing third party resources and data, and reliance on external dependencies, such as external web services, which add difficulty to reproducibility at a later time.

Currently, most of the approaches that address reproducibility of scientific workflows have focused either on their *physical preservation*, in which a workflow is conserved by packaging all of its components, so an identical replica is created and can be reused; or on *logical preservation*, in which the workflow and its components are described with enough information for others to reproduce a similar workflow in the future [9].

Although both, packaging and describing, play a vital role in supporting workflow re-use, alone they are not sufficient to effectively maintain reproducibility. On the one hand physical preservation is limited to recreating the packaged components and resources while it lacks a structured description about the workflow. Thus, it makes easy to *repeat* exactly the same execution, yet it is often not enough to *reproduce* the experiment with different parameters or input data. On the other hand logical preservation can provide detailed description of various levels of the workflow. It is still not enough, however, in the absence of the necessary tools and dependencies.

A need to integrate these two forms of preservation becomes increasingly apparent. That, combined with a portable description of the workflow, which can be used in different environments, and an automated workflow deployment mechanism has potential to significantly improve workflow reproducibility.

In this paper we present a framework designed to address the challenges mentioned earlier. The framework integrates features of both logical and physical preservation approaches. Firstly, using OASIS specification "Topology and Orchestration Specification for Cloud Applications" (TOSCA) [10] it allows a workflow description to include the top-level structure of the abstract workflow together with details about its execution environment. The description is portable and may

be used in automated deployment across different execution environments including the Cloud and a local VM.

Secondly, using Docker virtualisation and imaging our framework offers portable packaging of whole workflows and their parts. By integration with TOSCA, the packaging is automated, hence users are free from creating and managing Docker images. Additionally, our framework is built upon code repositories that natively support version control – crucial in tracking the evolution of workflows and their components over time.

We argue that these three elements: portable and comprehensive description, portable packaging and widely applied version control, play a fundamental role in maintaining reproducibility of scientific workflows over longer periods of time. They allowed us to build the framework which we present as the main contribution of this paper. We evaluate the framework using real scientific workflows developed in our previous projects to demonstrate that it can effectively realise its goal.

II. Background and Related Work

Workflow reproducibility and repeatability have been discussed in a number of studies, such as [5], [11], and are considered to be an essential part of the computational scientific method. As our approach to improving reproducibility of workflows is based on the TOSCA specification and Docker technology, in this section we present the three relevant areas.

A. Scientific Workflow Reproducibility

There have been various attempts proposed in the literature or as software tools to address repeatability and reproducibility of scientific workflows. As mentioned earlier, most of them follow one of two directions: (1) packaging the components of a workflow, known as physical preservation/conservation, or (2) describing a workflow and all its components, called logical preservation/conservation.

To implement packaging of workflows Chirigati et al. proposed ReproZip [12]. It tracks system calls during the execution of a workflow to capture the dependencies, data and configuration used at runtime, and to package them all together. Then the package can be used to re-execute the archived workflow invocation.

Other researcher to package workflows have used virtualization mechanisms, specifically the ability to save the state of a virtual machine as an image (VMI) [13], [14], [15]. The main advantage in using VMIs is that they allow the complete experimental workflow and environment to be easily captured and shared with other scientists [16]. However, the resulting images are large in size and costly to be publicly distributed [17]. And despite packaging mechanisms allowing workflows to be re-executed (i.e. allow repeatability), they usually do not convey a detailed and structured description of the entire computation, relevant dependencies and execution environments, which would help in understanding the package contents. Therefore, their ability to reproduce or even reuse a packaged workflow in other contexts (e.g. using different input data, parameters or execution environments) is often limited.

The logical preservation techniques focus on capturing all the details required to repeat and potentially reproduce scientific workflows. A notable example is myExperiment [18] which offers a web interface to support social sharing of workflows with computational description and visualizations of their components. myExperiment, as a general repository for workflows, contributes to the improvement of workflow reproducibility.

Santana-Perez et al. proposed in [9] a semantic-based approach to preserve workflows with their execution environment. They use a set of semantic vocabularies to specify the resources involved in the execution of a workflow. However, other studies have shown that sharing only the specifications of a workflow is not enough to ensure successful reproducibility [19].

Another technique of logical preservation is capturing the provenance information of the workflow results [20], [21]. Retrospective provenance encapsulates the exact trace of a past workflow execution, which can then help in its re-execution. Nevertheless, provenance usually describes only the abstract layer of a workflow because detailed traces of the use of execution environment (e.g. at the OS level) quickly become overwhelming.

More recently, Hasham et al. [22] presented a framework that captures information about Cloud infrastructure of a workflow execution and interlinks it with data provenance of the workflow. They propose workflow reproducibility by re-provisioning similar execution infrastructure using the Cloud provenance and then re-execution of the workflow. Although the approach enables the re-execution, it is unable to track and address changes to the original workflow.

Belhajjame et al. [19] proposed Research Objects as a preservation approach for scientific workflows. Research Objects can aggregate various types of data to enhance workflow reproducibility like: workflow specifications, description of workflow components and provenance traces. However, they do not include enough technical details about dependencies and the workflow execution environment to easily allow re-enactment.

The specification-based mechanisms provide various details that can help in understanding the workflow and its components. Yet, they are still insufficient when some of the required dependencies change or become unavailable, in which case the ability to reconstruct the same execution environment is lost. Therefore, the integration of workflow specification and description of its components alongside a portable packaging mechanism that facilitates sharing becomes fundamental.

B. Topology and Orchestration Specification for Cloud Applications

TOSCA is an OASIS specification for modeling a complete application stack, and automating its deployment and management in the Cloud. The main intent of TOSCA is to improve

the portability of Cloud applications in the face of the growing diversity of Cloud environments [23].

The specification defines a meta-model for describing both the structure and management of IT applications. The structure of an application, its components and the relationships between them are represented by a *Topology Template*. The components and their relationships are defined as *Node* and *Relationship Templates* instantiated from a predefined *Node* and *Relationship Types*. The types are reusable entities that can be used to construct new Topology Templates for different applications [24].

In our previous work [25] we proposed the use of TOSCA to describe the entire structure of a scientific workflow, together with all its components and specification of a host environment. By adopting TOSCA, we can turn workflows into reusable entities that include not only the description of a scientific experiment but also all details needed to deploy and execute them automatically. Therefore, we use TOSCA as the basis for the framework presented in this paper.

C. Reproducibility using Lightweight Virtualization

Container-based virtualization is a lightweight alternative to Virtual Machines. It is not new but Docker,[1] one of the recently developed tools for Linux systems, established a strong and open ecosystem that several Cloud providers support and promote in their offers. Importantly, Docker containers are portable and can run on different hosts, which makes them a suitable packaging tool to support the reproducibility of applications [26].

Similarly to a VMI, a Docker image is a file that includes an Operating System together with a set of relevant libraries, software packages, configuration and data. And it can later be used to create a container – a running instance of the system/application. That makes containers equally suitable to encapsulate and then re-execute scientific workflows. But the main attraction in using containers, when compared to Virtual Machines, is that images are smaller in size and starting a container is a few orders of magnitude faster than starting an VM. Therefore, in our work we integrated Docker images and containers in the deployment and reproducibly process.

Similarly to Virtual Machine hypervisors, Docker allows workflow applications along with all necessary dependencies to be encapsulated into a container image [27], [28]. But even if these approaches can offer a convenient mechanism to preserve workflows, they still lack a structured description of the aggregate. In addition, they are limited to packaged resources and dependencies, and lack flexibility to change the components or dependencies in an already packaged workflow.

III. Improving Workflow Reproducibility

Clearly, the complete reproducibility of a workflow is hard to achieve due to possible changes at various levels of software and hardware platforms used to run it. We can, however, significantly increase the degree of reproducibility by addressing the challenges discussed earlier. And our goal when designing our workflow reproducibility framework was to find ways in which we can effectively respond to these challenges.

[1]https://www.docker.com/

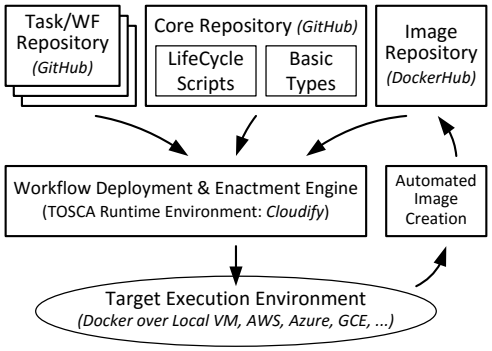

Fig. 1: The architecture of our workflow reproducibility framework.

A. The Framework Architecture

The proposed workflow reproducibility framework consists of four main components: the Core repository, a set of Workflow and Task repositories, the Image repository supported by the Automatic Image Creation (AIC) facility, and the workflow enactment engine (Fig. 1). The Core repository includes a set of common and reusable TOSCA elements such as NodeTypes and RelationshipTypes, and life cycle management scripts. They are a foundation for building tasks and workflows. The Workflow and Task repositories are used to store workflows and their components so they can be accessed during enactment and also shared and reused in designing new workflows. The Image repository contains workflow and task images that are used to improve reproducibility and also performance of workflow enactment. Images are automatically captured by the AIC. Finally, the workflow enactment engine is implemented by a TOSCA-compliant runtime environment.

To implement logical preservation we rely on the TOSCA specification which we previously adopted as a method to model portable workflows [25]. With TOSCA we can describe workflows not only at the abstract level but together with the complete software stack required to deploy and enact them. And it is portable because we can use a TOSCA-compliant runtime environment to automatically deploy and enact our workflows on different Cloud platforms or in a local VM.

To control changes that can affect a workflow and its components we use a version control platform. It gives us the ability to track the complete history of developmental changes of workflows and tasks. The version control platform supports also the Automatic Image Creation facility. The AIC uses Docker to implement physical preservation of workflows and greatly helps in building and management of image libraries.

Moreover, instead of building yet another workflow repository and yet another workflow engine, we define our framework on top of open platforms like GitHub and DockerHub. The former allows workflow and task source code to be stored

and maintained under version control, the latter can store workflows and tasks packaged as Docker images. Importantly, both platforms offer mechanisms which promote sharing and reuse.

B. The Framework in Use

To create a workflow the user needs to implement and model its essential components including: NodeTypes and tasks code. The NodeTypes are used to declare tasks and dependency libraries, they also refer to the task code – the actual software artifacts which will be deployed and executed.

Currently, to facilitate building new tasks and workflows we implement a set of basic NodeTypes and tasks which others can reuse. Additionally, our Core repository provides RelationshipTypes and life cycle management scripts that are common to all workflows. They define and implement basic workflow functionality like passing data between tasks, configuration of library dependencies, etc. Given all these components, the workflow can be encoded as a TOSCA ServiceTemplate. The template includes Node- and RelationshipTemplates that are instances of the types developed earlier; these templates represent tasks and task links, respectively.

Once the workflow ServiceTemplate has been prepared it can be deployed by a TOSCA-compliant runtime environment. Currently, we support Cloudify[2] but there are other options available like OpenTOSCA[3] and Alien4Cloud.[4] The enactment of workflows follows the structure embedded in the TopologyTemplate, a part of the ServiceTemplate that in a declarative way combines components and dependencies. Using the TopologyTemplate the runtime environment is able to infer the appropriate workflow execution plan.

IV. Workflow and Task Repositories

Since we have been using publicly available platforms like GitHub to maintain the Workflow and Task repositories, these repositories can remain under users' control. We provide our own repositories with a set of basic reusable workflow tasks and example workflows mainly to illustrate how the framework can support reproducibility. But primarily, the ecosystem of workflows and tasks will be grown by researchers and scientists who want to develop their own workflow applications.

The choice of source version control platforms, such as GitHub, to host repositories of workflows and tasks was not accidental. These platforms offer great tools to support sharing and communication. But more importantly, they allow code developers and users to keep track of the developmental changes, and that can directly help to improve repeatability and reproducibility.

Our approach works on the principle that each single workflow and workflow task is maintained in a separate code repository. That brings multiple benefits: repositories mark clear boundaries between components, they offer independent version control, allow for easy referencing and sharing, and additionally, provide branches and tags to implement strict control of workflow and task interface. With multiple repositories it's also easy to encapsulate auxiliary information, such as sample data and human readable description specific to each workflow and task, which help to maintain long-term reproducibility.

A. Repository Structure

A repository aggregates various artifacts with information and resources related to the workflow or task. These artifacts include: TOSCA-based descriptors, workflow/task-specific life cycle scripts, sample data, human readable description and the *one-click* deployment script. The key and mandatory artifact is a TOSCA-based descriptor. In the case of a workflow, it is a ServiceTemplate descriptor that encodes the structure of a workflow and references all the workflow components and life cycle scripts required for enactment. In the case of a task, the descriptor includes TOSCA NodeType that defines the task interface and refers to the actual task implementation code.

Other artifacts, although optional, are helpful to maintain reproducibility. For example, provided with sample data and the one-click deployment script users can easily test a workflow or task in their environment. The script starts a multi-step process which deploys the workflow together with basic dependencies such as Docker and Cloudify and then enacts it. Moreover, given a human readable description stored in a repository, users can better understand the purpose of the component and more easily use it. That also helps to recover from failures in the face of changes in the workflow or any of its dependencies.

The structures of a workflow and task repository are very similar to each other. This is because our tasks also include a simple test workflow descriptor and sample data which allow users to easily run a task and test whether it actually meets their requirements.

Usually, our repositories include two workflow descriptors that define the *single-* and *multi-container* configuration. The single-container workflows are executed within one Docker container, whereas in the multi-container configuration *each* task runs in its own container. The use of the single- or multi-container configuration has also impact on the kind of images that will be generated by our Automatic Image Capture facility. We discuss this aspect later in Section VI.

These two default configurations describe, however, only two extremes out of the range of possible workflow deployments. For more specific, advanced scenarios developers can create workflows that include containers which group together a subset of tasks, for example due to security reasons.

B. Interface Control via Branches and Tags

One of the major sources of workflow decay is changes in the components that a workflow is comprised of. In a living system changes are inevitable because the components – tasks, libraries and other dependency workflows – undergo continuous development. Yet to maintain reproducibility we

[2]http://getcloudify.org
[3]https://github.com/OpenTOSCA
[4]http://alien4cloud.org

cannot forbid changes at all. Instead, we need to control them, so they do not contribute to the decay.

The changes that occur naturally during workflow and task development can affect two layers: the interface and/or implementation of a component. By the workflow/task interface we consider the contract between the developer and user of a component. Specifically, it is the number and type of input data and properties that the workflow/task uses in processing but also the number and type of output data it produces.

Changes in the interface usually indicate some important modification to a component and need to be followed by changes in its implementation. Conversely, changes to the implementation only, if made carefully, are often merely improvements in the code which can remain unnoticed.

Since in our framework each component has been maintained in a separate repository, we can control these two types of changes effectively. We use repository branches to denote changes in the interface, and tags to indicate significant improvements in the implementation. Minor implementation changes are simple commit events in the repository which do not need any special attention. All that, supported by an effective way to reference a specific branch or tag offered by GitHub are enough to address the problem of changing components.

However, these mechanisms are not only important for our framework to maintain reproducibility of existing workflows but they are also crucial for users in creating new workflows. With repository branches users can easily see different flavours of a specific task or workflow and decide which one to use. On the other hand tags help users to see major improvements of a component or workflow. Tags also indicate to our framework when there is a need to create a new component image.

To illustrate the use of branching and tagging in practice we show later, in the Evaluation section, a development scenario of one of our test workflows.

V. Automated Workflow Deployment and Enactment

The model of describing workflows using TOSCA proposed in our previous work [25] is important because it not only supports logical preservation but also offers the ability to automatically deploy and enact our workflows. That facilitates repeatability and improves workflow reproducibility.

Currently, as a workflow engine we use Cloudify – a TOSCA-compliant runtime environment. To run a workflow, users need to clone its repository to a target machine in which they are going to run it. The repository includes sample data and the *one-click* deployment script. It is a simple script able to install the software stack required to run the workflow (Cloudify, Docker and some auxiliary tools) and then to submit the workflow to Cloudify with default configuration parameters.

The default configuration and sample data allow users to easily test the workflow. It is also a means to repeat the execution as well as a starting point to reproduce it. To repeat a workflow users can simply switch to a very specific version

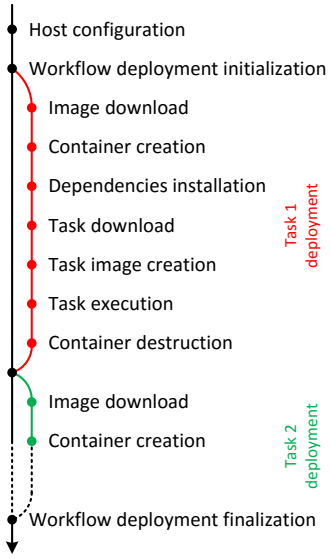

Fig. 2: Steps in automatic workflow deployment using the multi-container configuration.

of the workflow in the repository history and run the *one-click* deployment script. Then, they can modify the default configuration and provide their own data. They can also switch to the latest version of the workflow to validate the output or compare it with output generated by previous versions.

The TOSCA descriptor of a workflow is a declarative specification that includes all tasks, dependency libraries and task links embedded in the workflow ServiceTemplate. The template includes also dependency against the task execution environment which may be composed of one or more Docker containers and VMs. Apart from the declaration of tasks and libraries, the workflow ServiceTemplate encodes also the topology of the workflow. For scientific workflows, usually implemented as directed acyclic graphs, it is enough information so a linear workflow execution plan can be automatically inferred (Fig. 2). Cloudify follows the generated plan, and deploys and runs one task at a time.

Crucial to workflow enactment are life cycle management scripts. They implement deployment operations that each workflow and task needs to go through, such as: initialization of a shared space used to exchange data between tasks, provisioning of the host environment (a container), installation and configuration of library dependencies. As the majority of tasks follow a very similar pattern of deployment, we developed a set of common, reusable life cycle scripts and included them in the Core repository. Developers would refer to these scripts when building their own workflows and tasks.

VI. Automatic Image Capture

TOSCA-based descriptors are the fundamental element of our framework, partly because they are used to implement logical preservation, and partly because they allow workflows to be automatically deployed and enacted. But running workflows based only on these descriptors would end with significant

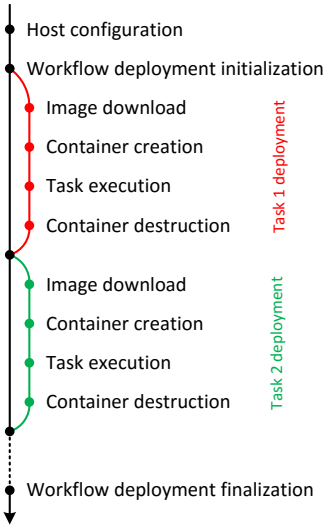

Fig. 3: Steps in automatic workflow deployment using the task images created by the AIC; cf. Fig. 2.

runtime overheads. The framework would repeat the same, sometimes long running, steps to deploy a task every time it was executed.

However, our framework is flexible enough to run the workflow and task deployment process using a variety of Docker images – starting from a pure OS image commonly available from DockerHub, to a specific user-defined image which includes some workflow/task dependencies, to a complete image that contains all of the required dependencies. If the image referred to in the workflow ServiceTemplate does not contain all the dependencies, they will be installed by the framework on-demand during workflow enactment. That automation simplifies the development cycle because users are not forced to manually prepare and manage task or workflow images before they can use a workflow.

Yet, to simplify the use of the framework even more we implemented the Automatic Image Capture facility. Using the Docker image manipulation operations, the AIC is able to create workflow and task images for the user automatically, so they can be deposited in a private or public Image Repository. Next time when a task is executed, instead of the complete deployment cycle, the framework will use the images captured earlier (Fig. 3). As shown later in the Evaluation section, that simplification can have very positive impact on runtime performance.

The workflows we implemented are usually described with two configuration options: single- and multi-container. That influences the way in which deployment and enactment of workflows is performed. But it also determines what image the AIC will create for the workflow. If the workflow uses the single-container configuration, the AIC will capture a single image that encapulates the whole workflow with all its components. Conversely, if the workflow uses the multi-container configuration, many smaller task images will be created. Both options have their advantages: the former imposes

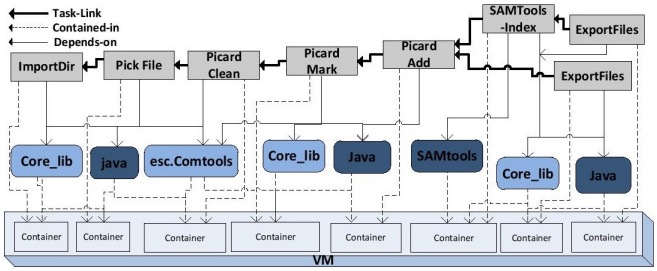

Fig. 4: The structure of the Sequence Cleaning workflow in multi-container configuration described in TOSCA.

less overhead in terms of storage and performance, whereas the latter promotes better reuse of task images and gives more flexibility if the workflow requires updates. Nonetheless, they support repeatability and reproducibility of workflows equally well.

Yet, to realise that goal images must be properly versioned. The AIC uses identifiers from the Image Repository and tags from the Workflow and Task Repositories to address this aspect. The workflow/task image identifier is generated based on the base Docker image identifier and the URL of a branch or tag of a workflow/task for which the image is built. That simple and *unique* mapping between code and image versions allows users to include only the *code* URL in their workflow ServiceTemplate, which is enough for the framework to fetch and use the correct image for a task or workflow. And in the case that the image does not yet exist, the workflow enactment will follow the full deployment cycle while the AIC will generate and deposit the relevant images for future use.

VII. Evaluation and Discussion

We describe the evaluation of our framework from three different angles. First, we present a set of experiments to show portability of the workflow description, so it can be enacted in different environments. Second, we show the benefit of using the AIC to reduce a workflow's runtime. Finally, we describe a scenario of workflow and task development to illustrate how the framework can maintain reproducibility in the face of component changes.

A. Repeatability on Different Clouds

The goal of this set of experiments was to re-enact a workflow, initially designed in a local development environment, on three different Clouds and a local VM. We ran the experiment for four different workflows which were previously designed in e-Science Central.[5] The workflows: *Neighbor Joining* (NJ), *Sequence Cleaning* (SC), *Column Invert* (CI) and *File Zip* (FZ) are different in terms of structure, dependency libraries they require and the number of tasks they include (11, 8, 7 and 3 tasks, respectively). As an example, Fig. 4 depicts the structure of the Sequence Cleaning workflow used in a NGS pipeline [29] and re-implemented using TOSCA.

[5]http://www.esciencecentral.co.uk

TABLE I: Basic details about the execution environments.

Environment	CPU Cores	RAM [GiB]	Disk space [GB]	Operating System
Local VM	1	3	13	Ubuntu 14.04
Amazon EC2	1	1	8	Ubuntu Srv 14.04
Google Cloud	1	3.75	10	Ubuntu Srv 14.04
Microsoft Azure	1	3.5	7	Ubuntu Srv 14.04

TABLE II: The average execution time (in minutes) for different workflows executed in different environments.

	Neighbour Join.		Column Invert		File Zip	
	Single	Multi	Single	Multi	Single	Multi
Devel. Env.	2.13	2.54	0.9	1.3	0.6	0.94
Amazon	1.74	2.27	0.66	1.18	0.5	0.84
Azure	2.52	3.86	1.35	2.1	1.23	1.38
Google	1.52	2.48	0.74	1.18	0.5	1.01
Local VM	1.65	2.5	1.03	1.37	0.53	1.03

To illustrate the potential of our framework in supporting repeatability and reproducibility and the value of the proposed workflow representation, each of the selected workflows was first developed, and then deployed and enacted in a local development environment. We recorded the execution time of that initial enactment which also automatically created a workflow or task Docker images.

To conduct the rest of the experiment, we cloned the workflow repositories in four different environments: a local VM, and Amazon AWS, Google Engine and Microsoft Azure Clouds. Finally, we re-executed workflows five times in each VM and collected results. The configuration of the VMs is presented in Table I.

Each workflow was used in two available configurations: single- and multi-container to show the overheads of running multiple task containers. The output data of the workflows were the same in all executions and the average execution times were similar. Fig. 5 shows a chart with the results for the SC workflow, whereas Table II includes the results for the other tested workflows.

The experimental results show that our scientific workflows can be re-enacted, producing the same outputs in similar runtime. They also illustrate a common development pattern in which developers build and test a workflow in their local environment and once it is ready they can share it with others via Workflow, Task and Image Repositories. Both the TOSCA representation and Docker packaging offer significant support for this pattern.

B. Automatic Image Capture for Improved Performance

As mentioned earlier, our framework is flexible enough to allow tasks and workflows to use pure OS images available from DockerHub or custom, predefined task/workflow images created by users or the AIC. By using a predefined image we can avoid the installation of dependency libraries and task artifacts required during workflow execution. And, as shown

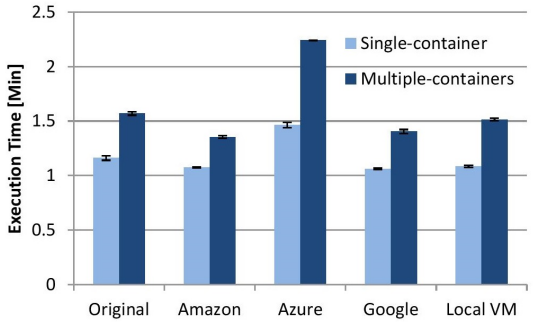

Fig. 5: The average execution time for Sequence Cleaning workflow executed in different environments.

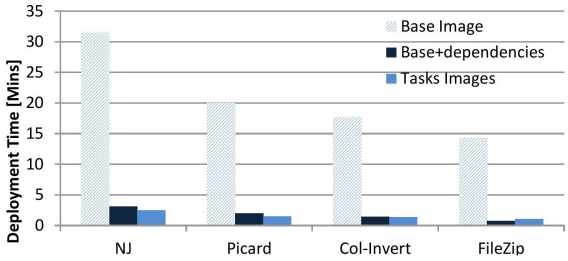

Fig. 6: The average execution time of test workflows using different task images.

previously in Fig. 3, that can reduce the number of deployment steps required in workflow enactment.

However, the elimination of some of the deployment tasks can have very positive impact on the runtime of workflows. To show it we prepared a set of experiments in which we ran our workflows using different images: the base image available on DockerHub, the base image with pre-installed dependency libraries and task images captured by the AIC. Fig. 6 depicts the average workflow execution time for four tested workflows.

Clearly, there was a significant overhead in using the base image from DockerHub. The main reason was the time required to install dependency libraries such as the Java Runtime Environment or, in the case of the NJ workflow, the Wine library.

The second and third option show small differences with slightly shorter execution for experiments which used images created by the AIC. That is because the AIC captures everything the task needs to run (according to the task's TOSCA descriptor), whereas the second option included only dependency libraries while the task artifacts were downloaded and installed on-demand.

The results explicitly show that from the performance perspective the use of pre-packaged images is the most effective option. However, from the user perspective the quickest and easiest is the use of the base images already available on DockerHub instead of building images manually. Our framework supports such flexibility for the cost of some overhead incurred by the initial execution of a workflow. The first run will involve the complete deployment cycle and creation of the

images, whereas any subsequent executions will benefit from that images and will run at full speed.

C. Reproducibility in the Face of Development Changes

One of the key factors that can reduce decay of our workflows is their ability to embrace changes that occur naturally during workflow and task development. These changes may affect mainly two layers: the input/output interface of a workflow or task, and their implementation.

In Fig. 7 we illustrate a *hypothetical* evolution scenario of the Sequence Cleaning workflow shown earlier in Fig. 4. The left side depicts the timeline of development events that occured in the scenario. It is accompanied by change trees from two repositories: the left tree represents the evolution of the workflow, the one on the right shows the evolution of one of the workflow tasks.

We start the analysis with the version of the SC workflow presented earlier and tagged as `v1` in Fig. 7 (event 1). By tagging we acknowledge that this version has been published, advertised and so may be used by others.

Now, let us imagine that a new requirement for our workflow appeared (event 2) – users of the workflow want to save storage space by compressing the workflow output files. In response to that, the developers created a new `Zip` task (cf. the right version tree) and wanted to add it to the workflow. Note, however, that changing the type of outputs generated by the workflow is a change of its interface. For example, it would likely break any external application that has used uncompressed outputs provided by version `v1`. Thus, before we can add the `Zip` task to the workflow we need to create a new branch, named `zipped` in the figure (event 3).

The `zipped` branch of the workflow refers to the `Zip/master` branch of the task. By default such a reference means that the workflow depends on the latest tagged version of the task coming from that branch. This is convenient because as the task implementation is improved over time, the `zipped` workflow will use a task's latest tagged version (including `v1.1`). In this way workflows are updated automatically without the need to change them when only implementation improvements are made to the tasks. However, if strict workflow repeatability is required, the reference to the `Zip` task would include a specific tag. That would prevent the automatic update of such a workflow.

Next, event (4) denotes a new release of the `Java` library used by some tasks in the workflow. In our hypothetical scenario the new version of the library has improved performance and many errors fixed. Thus, the event is a signal for us to update the workflow as soon as possible. That change is compatible with the previous version of the workflow and so we do not need to create a new branch. Instead, we merge in the changes from `master` to the `zipped` branch, so that both branches can benefit from the updated library.

After adding the `Zip` task and updating the `Java` library we also tag and announce new, improved versions of our workflow (event 5). Specifically, `SampleCleaning/v2` runs faster and produces smaller outputs which is of great value to the users.

Event (6) marks the arrival of yet another requirement – users want the outputs of the workflow to be encrypted to avoid leakage of patients' raw genomic data. That requires, however, some improvements in the `Zip` task, including changes to the underlying tool used to compress the data.

After running some tests it appeared that the new zip tool has much better performance, and so we quickly decided to swap the old implementation with the new tool and tag the task `v1.1`. Note that this simple act of tagging a version causes an automatic update of all workflows that rely on that branch. Therefore, from now on the `SampleCleaning.v1.1` and `.v2` workflows will use the updated implementation of the `Zip` task.

Continuing with the task update, we create a new `password` branch in the task repository (event 7). This new branch is needed due to the changes in the task's input interface – the new version has the extra `password` input property. But the use of encryption is optional, so to limit the number of branches we decided to discontinue the previous version of the `Zip` task and tag branch `master` as `deprecated` (event 8). That indicates to users that they should use other branches of the task in their new workflows. Nonetheless, the old version will need to remain in the repository because others may still use workflow `SampleCleaning.v2` which relies on the `Zip/master` branch.

As proactive workflow developers we noticed that the `master` branch of the `Zip` block has been deprecated and so we decided to update the reference in the `zipped` version of the workflow to the active `password` branch. Note that this update does not require a new branch because the use of encryption in the `Zip` task is optional. Thus, the workflow's input and output interface can remain the same (event 9).

The new branch is created later (event 10) when the `password` property is exposed to end users as the workflow input property. We want the users to be able to set a custom password for the output data and that requires a change in the workflow interface which, in turn, requires a new branch.

The presented hypothetical evolution shows very common patterns in the development of workflows and their components – changes can occur at different layers of workflow and tasks. However, by means of separate task and workflow repositories, and conscious tagging and branching of their code, we can maintain all workflow versions in the working state and also ensure that their evolution does not break external applications that rely on them.

VIII. Conclusions and Future Work

Reproducibility is a crucial requirement for scientific experiments, enabling them to be verified, shared and further developed. Therefore, workflow reproducibility should be an important requirement in e-Science. In this paper we presented a design and prototype implementation of a framework that supports repeatability and reproducibility of scientific workflows. It combines two well-known techniques: logical and

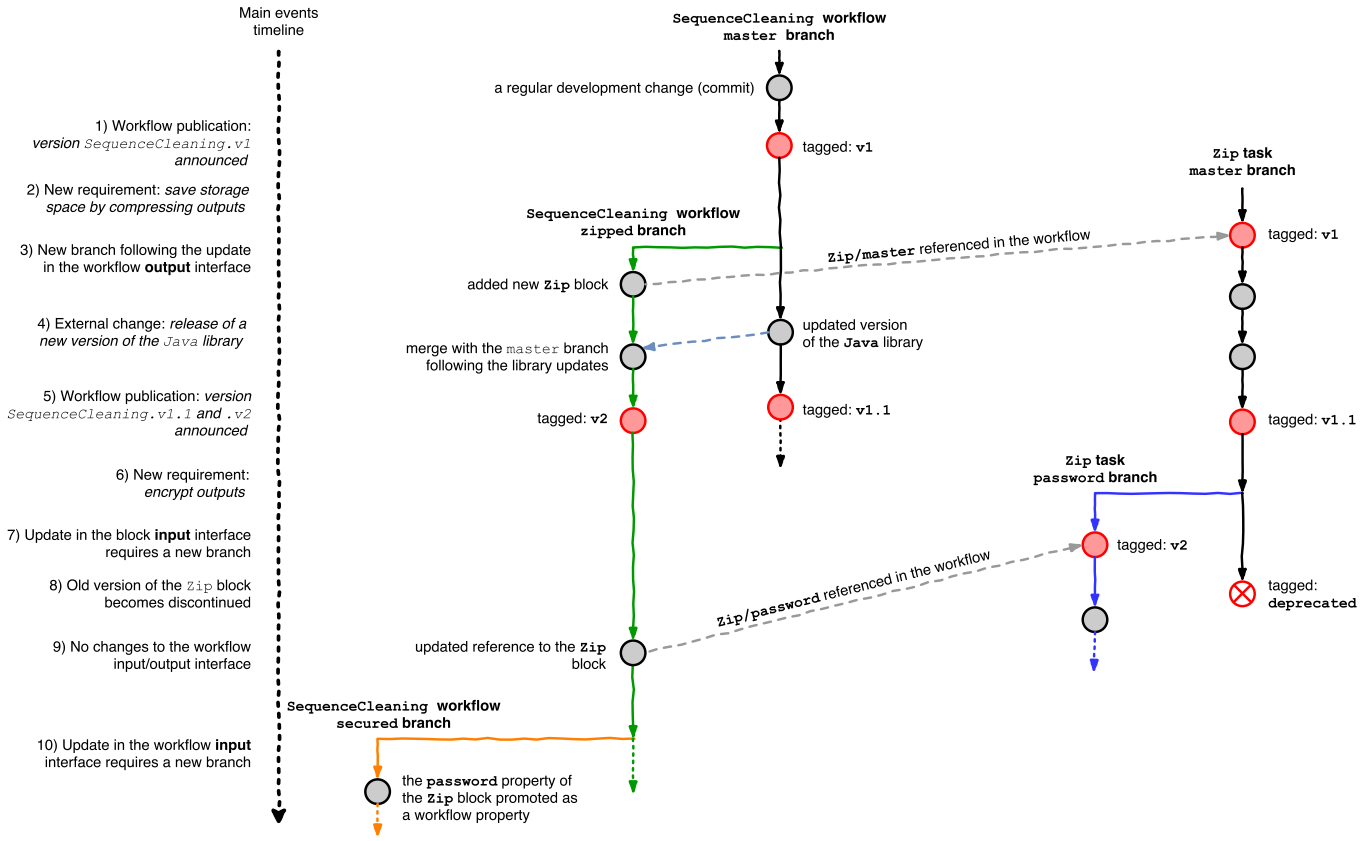

Fig. 7: A hypothetical evolution of the `Sequence Cleaning` workflow.

physical preservation. To implement the logical preservation technique we use the TOSCA specification as a means to describe workflows in a standardised way. To realise physical preservation we use lightweight virtualisation which allows us to package workflows, tasks and all their dependencies as Docker images.

Moreover, our framework uniquely combines software repositories to manage versioning of source code, an automated workflow deployment tool that facilitates workflow enactment and reuse, and automatic image creation to improve performance. They all significantly increase the degree of workflow reproducibility. And although, currently, our framework does not capture *retrospective* provenance traces, which has been left for future work, the proposed TOSCA-based workflow descriptors may be considered to be a detailed *prospective* provenance document. They describe the high-level structure of the workflow, which might also be encoded using, for example, the ProvONE specification,[6] together with all details to recreate the complete software stack needed for deployment and enactment.

Still, however, a considerable part of reproducibility is in the hands of workflow developers: scientists and researchers who will use our tools. Only with their help and dedication can workflows be adequately described, have sample input and configuration data to facilitate testing, and be properly versioned with branches and tags indicating major development events. Our framework merely makes these tasks easier.

As for the future, the presented work opens a variety of interesting research avenues. We plan to add a facility to capture retrospective provenance information for workflows and tasks that could complement the history of their development. We consider implementing support for large-scale, distributed workflow enactment. Finally, we also plan to investigate to what extent our framework can model legacy workflows designed in other scientific workflow management systems like Pegasus and Taverna.

[6]The latest draft of the ProvONE specification from May 2016 is available at: http://jenkins-1.dataone.org/jenkins/view/Documentation%20Projects/job/ProvONE-Documentation-trunk/ws/provenance/ProvONE/v1/provone.html

IX. Acknowledgement

This work was partially supported by EPSRC grant no. EP/N01426X/1 in the UK.

References

[1] E. Deelman, D. Gannon, M. Shields, and I. Taylor, "Workflows and e-Science: An overview of workflow system features and capabilities," *Future Generation Computer Systems*, vol. 25, no. 5, pp. 528–540, May 2009.

[2] B. Liu, B. Sotomayor, R. Madduri, K. Chard, and I. Foster, "Deploying Bioinformatics Workflows on Clouds with Galaxy and Globus Provision," *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pp. 1087–1095, Nov. 2012.

[3] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682–692, Mar. 2013.

[4] H. Meng, R. Kommineni, Q. Pham, R. Gardner, T. Malik, and D. Thain, "An invariant framework for conducting reproducible computational science," *Journal of Computational Science*, vol. 9, pp. 137–142, 2015.

[5] J. Zhao, J. M. Gomez-Perez, K. Belhajjame, G. Klyne, E. Garcia-Cuesta, A. Garrido, K. Hettne, M. Roos, D. De Roure, and C. Goble, "Why workflows break Understanding and combating decay in Taverna workflows," in *2012 IEEE 8th International Conference on E-Science*. IEEE, Oct. 2012, pp. 1–9.

[6] J. Goecks, A. Nekrutenko, and J. Taylor, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences." *Genome biology*, vol. 11, p. R86, 2010.

[7] A. Banati, P. Kacsuk, and M. Kozlovszky, "Four level provenance support to achieve portable reproducibility of scientific workflows," in *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, no. May. IEEE, May 2015, pp. 241–244.

[8] J. Freire, P. Bonnet, and D. Shasha, "Computational reproducibility: state-of-the-art, challenges, and database research opportunities," *Proceedings of the 2012 ACM SIGMOD ...*, pp. 593–596, 2012.

[9] I. Santana-Perez, R. F. da Silva, M. Rynge, E. Deelman, M. S. Pérez-Hernández, and O. Corcho, "Reproducibility of execution environments in computational science using Semantics and Clouds," *Future Generation Computer Systems*, 2016.

[10] O. Standard, "Topology and Orchestration Specification for Cloud Applications version 1.0," pp. 1–114, 2013.

[11] S. Arabas, M. R. Bareford, L. R. De Silva, I. P. Gent, B. M. Gorman, M. Hajiarabderkani, T. Henderson, L. Hutton, A. Konovalov, L. Kotthoff, C. Mccreesh, M. a. Nacenta, R. R. Paul, K. E. J. Petrie, A. Razaq, D. Reijsbergen, and K. Takeda, "Case Studies and Challenges in Reproducibility in the Computational Sciences," pp. 1–14, 2014.

[12] F. Chirigati, D. Shasha, and J. Freire, "ReproZip : Using Provenance to Support Computational Reproducibility," *USENIX Workshop on the Theory and Practice of Provenance*, 2013.

[13] V. Stodden, F. Leisch, and R. D. Peng, *Implementing reproducible research*. CRC Press, 2014.

[14] B. Howe, "Virtual Appliances, Cloud Computing, and Reproducible Research," *Computing in Science & Engineering*, vol. 14, no. 4, pp. 36–41, Jul. 2012.

[15] F. Jiang, C. Castillo, C. Schmitt, A. Mandal, P. Ruth, and I. Baldin, "Enabling workflow repeatability with virtualization support," *Proceedings of the 10th Workshop on Workflows in Support of Large-Scale Science - WORKS '15*, pp. 1–10, 2015.

[16] O. Spjuth, M. Dahlö, F. Haziza, A. Kallio, E. Korpelainen, and E. Bongcam-Rudloff, "BioImg.org: A Catalog of Virtual Machine Images for the Life Sciences," *Bioinformatics and Biology Insights*, no. Vmi, p. 125, Sep. 2015.

[17] P. Bonnet, S. Manegold, M. Bjørling, W. Cao, J. Gonzalez, J. Granados, N. Hall, S. Idreos, M. Ivanova, R. Johnson, D. Koop, T. Kraska, R. Müller, D. Olteanu, P. Papotti, C. Reilly, D. Tsirogiannis, C. Yu, J. Freire, and D. Shasha, "Repeatability and workability evaluation of sigmod 2011," *SIGMOD Rec.*, vol. 40, no. 2, pp. 45–48, Sep. 2011.

[18] C. a. Goble, J. Bhagat, S. Aleksejevs, D. Cruickshank, D. Michaelides, D. Newman, M. Borkum, S. Bechhofer, M. Roos, P. Li, and D. de Roure, "myExperiment: A repository and social network for the sharing of bioinformatics workflows," *Nucleic Acids Research*, vol. 38, no. May, pp. 677–682, 2010.

[19] K. Belhajjame, J. Zhao, D. Garijo, M. Gamble, K. Hettne, R. Palma, E. Mina, O. Corcho, J. M. Gómez-Pérez, S. Bechhofer, G. Klyne, and C. Goble, "Using a suite of ontologies for preserving workflow-centric research objects," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 32, pp. 16–42, 2015.

[20] P. Missier, S. Woodman, H. Hiden, and P. Watson, "Provenance and data differencing for workflow reproducibility analysis," *Concurrency Computation Practice and Experience*, no. October, 2013.

[21] D. McGuinness, J. Michaelis, L. Moreau, O. Hartig, and J. Zhao, "Provenance and Annotation of Data and Processes," *Ipaw*, vol. 5272, pp. 78–90–90, 2008.

[22] K. Hasham, K. Munir, R. McClatchey, and J. Shamdasani, "Re-provisioning of Cloud-Based Execution Infrastructure Using the Cloud-Aware Provenance to Facilitate Scientific Workflow Execution Reproducibility," in *Cloud Computing and Services Science*, 2016, pp. 74–94.

[23] Tobias Binz; Gerd Breiter; Frank Leymann; Thomas Spatzier, "Portable Cloud Services Using TOSCA," pp. 80–84, 2012.

[24] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, *TOSCA: Portable Automated Deployment and Management of Cloud Applications*, A. Bouguettaya, Q. Z. Sheng, and F. Daniel, Eds. New York, NY: Springer New York, 2014.

[25] R. Qasha, J. Cala, and P. Watson, "Towards Automated Workflow Deployment in the Cloud Using TOSCA," in *2015 IEEE 8th International Conference on Cloud Computing*. IEEE, Jun. 2015, pp. 1037–1040.

[26] D. Merkel, "Docker: lightweight Linux containers for consistent development and deployment," p. 2, 2014.

[27] R. Chamberlain and J. Schommer, "Using Docker to support Reproducible Research (submission to WSSSPE2)," pp. 1–4, 2014.

[28] C. Boettiger, "An introduction to Docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, Jan. 2015.

[29] J. Cała, E. Marei, Y. Xu, K. Takeda, and P. Missier, "Scalable and efficient whole-exome data processing using workflows on the cloud," *Future Generation Computer Systems*, Jan. 2016. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0167739X16000030